Self-Timed FIFO:
An Exercise in Compiling Programs into VLSI Circuits

Alain J. Martin

Computer Science Department
California Institute of Technology

5211:TR:86

The research described in this paper was sponsored by
the DEfense Advanced Research Projects Agency, ARPA Order No. 3771,
and monitored by the Office of Naval Research
under contract number N00014-79-C-0597

published in
IFIP WC 10.2 International Working Conference
on "From HDL Descriptions to Guaranteed Correct Circuit Designs",
Grenoble, France 9-11 September 1986. D. Borrione (ed)

Self-timed FIFO:
An Exercise in Compiling Programs into VLSI Circuits

Alain J. Martin

Computer Science Department
California Institute of Technology

5211:TR:86

SELF-TIMED FIFO: AN EXERCISE IN COMPILING PROGRAMS INTO VLSI CIRCUITS

Alain J. MARTIN

Computer Science Department, California Institute of Technology,
PASADENA, CA 91125, USA†

A method for compiling a high-level description of a computation (a set of communicating processes) into a self-timed VLSI circuit is explained with an example: the construction of a self-timed FIFO element. The method essentially relies on the four-phase handshaking expansion of the communication actions. The program of each process is compiled into a set of "production rules" from which all explicit sequencing has been removed. By matching the production rules to those describing the semantics of the VLSI-operators (and-gate, or-gate, C-element, arbiter, etc.), the programs are identified with networks of operators. We show how the different heuristics that the method allows lead to different circuits. In particular, the example illustrates the trade-offs between simplicity and efficiency of the circuits.

1. INTRODUCTION

We have developed a method for "compiling" a high-level description of a computation (a set of communicating processes) into a self-timed VLSI circuit. Self-timed [8] (or delay-insensitive [9]) circuits are sequential circuits in which the sequencing is enforced entirely by communication mechanisms. No clock signals are used, and no assumption is made on the delays in operators and wires except that the delays are finite. The advantages of self-time circuits are many: First, with the increasing size of circuits, it becomes more and more difficult to distribute safely a clock signal across a chip. Second, clocked circuits rely on worst-case assumptions on the timing behavior of the components, which decreases their performances. Third, with no restriction on the length of wires, layout is facilitated.

In the method we propose, the computation is initially described as a set of communicating processes in the notation of [3], which is somewhat similar to C.A.R. Hoare's CSP [2]. This first description is the reference solution,

† The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and was monitored by the Office of Naval Research under contract number N00014-792-C-0597

which has to be proved correct. The program is then compiled into a delay-insensitive circuit by applying a series of semantics-preserving transformations. Hence the circuit obtained is correct by construction: all semantic properties that can be proved of the program hold for the circuit as well.

The compilation is systematic and essentially relies on the four-phase handshaking implementation of communication actions. The program of each process is compiled into a set of "production rules" from which all explicit sequencing has been removed. By matching these production rules to those describing the semantics of the VLSI-operators (and-gate, or-gate, C-element, arbiter, etc.), the programs are identified with networks of operators, i.e., self-timed circuits.

The method has been applied to a whole spectrum of problems, some of them quite difficult, like distributed mutual exclusion [4] and fair arbitration [5]. The results are far beyond our expectations. For most circuits, especially complex ones, the compiled circuits are superior to their "hand-designed" counterparts, i.e. they are simpler and use fewer operators, in particular state-holding operators. A general description of the method can be found in [4] and [6].

As an exercise in applying the method, we will construct circuits corresponding to a self-timed FIFO-element. We will see how the different alternatives that the method allows lead to different solutions. We first present the program notation and the VLSI operators that constitute the "object code". We then describe the four steps of the compilation and illustrate the method by constructing different versions of the FIFO-element.

2. THE PROGRAM NOTATION

Sequential part

For the sequential part of the algorithm, we use a subset of Edsger W. Dijkstra's guarded command language [1], with a slightly different syntax. We will give only a very informal definition of the semantics of the constructs used.

i) $b\uparrow$ stands for $b :=$ **true**, $b\downarrow$ stands for $b :=$ **false**.

ii) The execution of the *selection* command $[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]$, where G_1 through G_n are Boolean expressions, and S_1 through S_n are program parts, (G_i is called a "guard", and $G_i \rightarrow S_i$ a "guarded command") amounts to the execution of an arbitrary S_i for which G_i holds. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \ldots \vee G_n)$ holds.

iii) Besides the usual sequencing operator—the semi-colon—, we introduce a weaker sequencing operator—the comma—. For atomic actions x and y, "x,y" stands for the execution of x and y in any order.

iv) $[G]$ where G is a Boolean, stands for $[G \rightarrow \mathbf{skip}]$, and thus for "wait until G holds". (Hence, "$[G]$; S"and $[G \rightarrow S]$ are equivalent.)

v) $*[S]$ stands for "repeat S forever".

vi) From ii) and iii), the operational description of the statement $*[[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]]$ is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds".

Communicating processes

A concurrent computation is described as a set of processes composed by the usual parallel composition operator $\|$. Processes communicate with each other by communication actions on channels; they do not share variables. When no messages are transmitted, communication on a channel is reduced to synchronization signals. The name of the channel is then sufficient for identifying a communication action.

If two processes $p1$ and $p2$ share a channel named X in $p1$ and Y in $p2$, at any time the number of completed X-actions in $p1$ equals the number of completed Y-actions in $p2$. In other words, the completion of the n-th X-action "coincides" with the completion of the n-th Y-action. If, for example, $p1$ reaches the n-th X-action before $p2$ reaches the n-th Y-action, the completion of X is suspended until $p2$ reaches Y. The X-action is then said to be *pending*. When thereafter $p2$ reaches Y, both X and Y are completed. The predicate "X is pending" is denoted $\mathbf{q}X$. If, for an arbitrary command A, $\mathbf{c}A$ denotes the number of completed A-actions, the semantics of a pair (X,Y) of communication commands is expressed by the two axioms:

$$\mathbf{c}X = \mathbf{c}Y \qquad (A1)$$

$$\neg\mathbf{q}X \vee \neg\mathbf{q}Y. \qquad (A2)$$

Probe

Instead of the usual selection mechanism by which a set of pending communication actions can be selected for execution, we provide a general Boolean command on channels, called the *probe*. In the original definition given in [3], the probe command $\overline{X}$ in process $p1$ has the same value as $\mathbf{q}Y$. Here,

we use a weaker definition, namely:

$$\overline{X} \Rightarrow \mathrm{q}Y$$
$$\mathrm{q}Y \Rightarrow \diamond\overline{X},$$

where $\diamond P$ means P *holds eventually.* For example, a construct of the form

$$[\overline{X} \rightarrow X \mid \overline{Z} \rightarrow Z]$$

can be informally interpreted as "if a communication action is pending at the other end of channel X, fire X; if a communication action is pending at the other end of channel Z, fire Z".

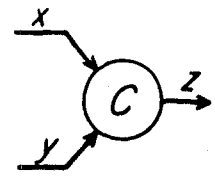
3. THE "OBJECT CODE"

The set of operators with which we build circuits is not unique. In this introduction, we will use the simple set consisting of *and*, *or*, *exclusive-or*, *C-element*, *enabled C-element*, *wire*, and *fork*. Each operator is described by a set of production rules. A production rule is similar to a guarded command, and we shall therefore use a similar syntax. There are, however, important semantic differences. Consider the production rule $G \mapsto S$:

- S is either a simple assignment or of the form "$s1, s2$" where $s1$ and $s2$ are each a simple assignment.
- If G holds, the correct execution of S is guaranteed only if G remains invariantly **true** until the completion of S. We say that G must be *stable.*
- Unlike the guarded commands of a selection or a repetition, the mutual exclusion among the different production rules of a set is not guaranteed automatically. It has to be enforced by the semantics of the program.
- If stability of the guards and mutual exclusion among guards are guaranteed, the production rule set PRS is semantically equivalent to the repetition $*[[GCS]]$, where GCS is the guarded command set syntactically identical to PRS. The descriptions of the operators used in this paper in terms of their production rules and their logic symbols are as follows.

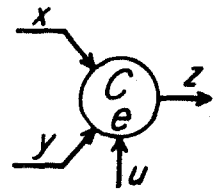
The C-element:

$$(x,y)\ \underline{C}\ z \equiv\ x \wedge y \mapsto z\uparrow$$
$$\neg x \wedge \neg y \mapsto z\downarrow$$



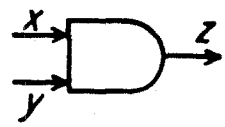
The enabled C-element:

$$(x,y;u)\ \underline{eC}\ z \equiv\ x \wedge y \wedge u \mapsto z\uparrow$$
$$\neg x \wedge \neg y \wedge u \mapsto z\downarrow$$



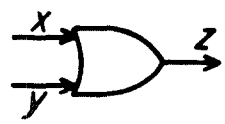
The "and":

$$(x,y)\ \underline{\wedge}\ z \equiv\ x \wedge y \mapsto z\uparrow$$
$$\neg x \vee \neg y \mapsto z\downarrow$$



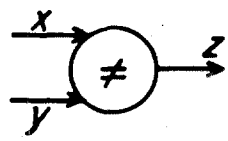
The "or":

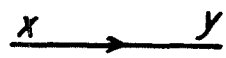
$$(x,y)\ \underline{\vee}\ z \equiv\ x \vee y \mapsto z\uparrow$$
$$\neg x \wedge \neg y \mapsto z\downarrow$$



The "exclusive-or":

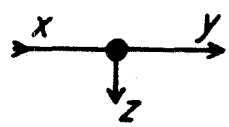
$$(x,y)\ \underline{Xor}\ z \equiv\ x \neq y \mapsto z\uparrow$$
$$x = y \mapsto z\downarrow$$





The wire:

$$x\ \underline{w}\ y \equiv\ x \mapsto y\uparrow$$
$$\neg x \mapsto y\downarrow$$



The fork:

$$x\ \underline{f}\ (y,z) \equiv\ x \mapsto y\uparrow, z\uparrow$$
$$\neg x \mapsto y\downarrow, z\downarrow$$

Any input or output variable of an operator may be negated. In particular, a wire with its input or its output negated—but not both—is an inverter. A negated input or output is represented in the figures by a small circle on the corresponding line.

4. THE COMPILATION METHOD

Process Decomposition

The first step of the compilation, called "process decomposition", consists in replacing a process by several semantically equivalent processes. The purpose of the decomposition is to obtain a process representation of the program in which the right-hand side of each guarded command is a straight-line program, i.e., consists only of simple assignments and communication commands, composed by semi-colons and commas.

Decomposition rule: *A process P containing an arbitrary program part S is semantically equivalent to two processes $P1$ and $P2$, where $P1$ is derived from P by replacing S with a communication action C on the newly introduced channel (C, D) between $P1$ and $P2$, and $P2 \equiv *[[\overline{D} \rightarrow S; D]]$.*

For example, a process P of the form

$$P \equiv *[S0; S1; S2]$$

can be replaced by the sematically equivalent program $(P1 \| P2)$, with

$$P1 \equiv *[S0; C; S2]$$

$$P2 \equiv *[[\overline{D} \rightarrow S1; D]].$$

Observe that the above decomposition does not introduce concurrency. Although $P1$ and $P2$ are potentially concurrent processes, they are never active concurrently: $P2$ is activated from $P1$, much as a procedure or a coroutine would be. The only purpose of this transformation is to simplify the structure of each command. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program.

Handshaking Expansion

The implementation of communication, called "handshaking expansion", replaces each channel by a pair of wire-operators and each communication action by its implementation. Channel (X, Y) is implemented by the two wires $(xo \ \underline{w} \ yi)$ and $(yo \ \underline{w} \ xi)$.

If X belongs to process $p1$ and Y to process $p2$, xo and xi belong to $p1$, and yo and yi belong to $p2$. Initially, xo, xi, yo, and yi—which we will call the "handshaking variables of (X, Y)"—are **false**. Assume that the program has been proved to be deadlock-free and that we can identify a pair of matching actions X and Y in $p1$ and $p2$ respectively. We replace X and Y by the sequences U_x and U_y respectively, with:

$$U_x \equiv xo\uparrow;\ [xi]$$
$$U_y \equiv [y_i];\ yo\uparrow.$$

Unfortunately, when the communication terminates, all handshaking variables are **true**. Hence, we cannot implement the next communication with

U_x and U_y. However, the complementary implementation can be used for the next matching pair, namely:

$$D_x \equiv xo\downarrow;\ [\neg xi]$$
$$D_y \equiv [\neg yi];\ yo\downarrow.$$

The solution consisting in alternating U_x and D_x as an implementation of X, and U_y and D_y as an implementation of Y is essentially the so-called "two-phase handshaking", or "two-cycle signaling". But it is in general not possible to determine syntactically which X- or Y-actions are following each other in an execution. In such cases, two-phase handshaking implementations require testing the current value of the variables. In this paper, we shall use a simpler but less efficient solution known as "four-phase handshaking", or "four-cycle signaling".

In a four-phase handshaking protocol, all X-actions are implemented as "$U_x; D_x$" and all Y-actions as "$U_y; D_y$". Observe that the D-parts in X and Y introduce an extra communication between the two processes whose only purpose is to reset all variables to **false**. The synchronization introduced by this extra communication is unnoticeable since the immediately preceding communication implemented by U_x and U_y sees to it that both processes reach a matching D_x and D_y "at the same time".

Both protocols have the property that for a matching pair (X, Y) of actions, the implementation is not symmetrical in X and Y. One action is called *active* and the other one *passive*. The four-phase implementation with X active and Y passive is:

$$X \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \tag{1}$$
$$Y \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow. \tag{2}$$

When no action of a matching pair is probed, the choice of which one should be active and which one passive is arbitrary, but a choice has to be made. The choice can be important for the composition of identical circuits. A simple rule is that for a given channel (X, Y), all actions at one side are active and all actions at the other side passive. If $\overline{X}$ is used, all X-actions are passive—with the obvious restriction that $\overline{Y}$ cannot be used in the same program.

The implementation of the probe is simply:

$$\overline{X} \equiv xi$$
$$\overline{Y} \equiv yi. \tag{3}$$

Given our definition of suspension, the proof that this implementation of the probe fulfils the definition of Section 2 is straightforward and is omitted.

A probed communication action $\overline{X} \rightarrow \ldots X$ is implemented:

$$xi \rightarrow \ldots xo\uparrow;\ [\neg xi];\ xo\downarrow.$$

Basic properties

The following properties of the handshaking protocol play an important role in the compilation method.

Property 1: *For the pair of wires* $(xo\ \underline{w}\ yi)$ *and* $(yo\ \underline{w}\ xi)$, *used together as in (1) and (2), and all variables* **false** *initially, the following sequence of transitions is guaranteed to occur if the system is deadlock-free:*

$$*[xo\uparrow;\ yi\uparrow;\ yo\uparrow;\ xi\uparrow;\ xo\downarrow;\ yi\downarrow;\ yo\downarrow;\ xi\downarrow]. \tag{4}$$

Hence, the following postconditions hold:

$$\begin{array}{l} xo\uparrow\{\diamond xi\} \\ xo\downarrow\{\diamond\neg xi\} \\ yo\uparrow\{\diamond\neg yi\} \end{array} \tag{5}$$

In other words, if the system is deadlock-free, the handshaking protocol guarantees that once $xo\uparrow$ has been completed, xi holds eventually. And similarly for $xo\downarrow$ and $yo\uparrow$.

Property 2: *Consider the handshaking expansion of a program p according to (1), (2), and (3). Provided that the cyclic order of the four handshaking actions of a communication command is respected, the last two actions of this command—the two actions of* D_x *or* D_y*—can be inserted at any place in p without invalidating the semantics of the communication involved. However, modifying the order of these two actions relatively to other actions of p may introduce deadlock.*

Property 2 is a direct consequence of the way in which we have introduced the sequences D_x and D_y. In this paper, we will ignore the deadlock issue when we re-order handshaking actions.

5. FOUR-PHASE FIFO-ELEMENT

A FIFO-element is a process—say, p—communicating with its left-hand neighbor by channel L and with its right-hand neighbor by channel R.

For instance, p receives a value from its left-hand neighbor by the input command $L?x$ and sends the received value to its right-hand neighbor by the output action $R!x$, as follows:

$$p \equiv *[\, L?x; R!x \,].$$

For the time being, let us ignore the transmission of values over the channels and let us concentrate on implementing the simpler program:

$$p \equiv *[\, L; R \,].$$

The program to be compiled is so simple that, a priori, we see no reason for using process decomposition. (We will see later that, even in this simple case, process decomposition can be useful.) We choose to implement communication commands L and R by four-phase handshaking, and, in view of our intention to compose several of these elements, we choose L to be passive and R active. This leads to the handshaking expansion of p:

$$*[[li];\ lo\uparrow;\ [\neg li];\ lo\downarrow;\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri]].$$

Because of the cyclic nature of the program, and because all variables are initialized to **false**, the above program is equivalent to

$$*[[\neg ri];\ [li];\ lo\uparrow;\ [\neg li];\ lo\downarrow;\ ro\uparrow;\ [ri];\ ro\downarrow]. \qquad (6)$$

6. PRODUCTION-RULE EXPANSION

The next step is to compile the handshaking expansion of the program into a set of production rules from which all explicit sequencing has been removed. By matching these production rules to the ones describing the semantics of operators, the programs can be identified with networks of operators. We use the compilation of p to illustrate the different steps of the expansion.

We start with the production-rule set syntactically derived from the program. In the case of p, it is the set derived from (6), namely:

$$\begin{aligned} \neg ri \wedge li &\mapsto lo\uparrow \\ \neg li &\mapsto lo\downarrow \\ \neg lo &\mapsto ro\uparrow \\ ri &\mapsto ro\downarrow . \end{aligned}$$

The execution of a production rule is called *effective* if it changes the value of a variable. Otherwise, it is called *vacuous*. We ignore vacuous executions of production rules. For each guarded command of the program, the production rule set representation is semantically equivalent to the program representation if and only if the order of execution of effective production rules is the same as the order of the corresponding transitions in the program—we call it the *program order*. (As a clue to the reader we list the production rules of a set in program order.)

In general, we have to strengthen the guards of some rules to enforce execution in program order. This is the case in our example: Since $\neg lo$ holds initially, the third production rule can be executed first if we don't strengthen the guards. Because all handshaking variables of L are back to **false** when L is completed, we cannot find a guard for the transition $ro\uparrow$. (Hence, the transitions following a semi-colon that can be identified with a semi-colon of the original program are likely to be difficult to deal with.)

One technique for solving this problem is to use the possibility of shuffling any of the last two actions of the four-phase expansion of a communication command as a consequence of Property 2. Of course, the shuffle must maintain the cyclic order of the four actions. The other technique consists in introducing a state variable to identify uniquely the state in which a certain transition is to take place.

In this exercise, we show that the different circuits for the four-phase FIFO correspond to the different ways to apply those two techniques. We first apply different shufflings of the handshaking actions. We will observe that more shuffling leads to simpler circuits and less shuffling to a "quicker return linkage". We start with the maximum shuffling and end with the quickest return linkage (no shuffling), which corresponds to an implementation with a state variable .

7. MAXIMUM SHUFFLING

The maximum shuffling that still maintains the order between the first half of the handshaking of L and the first half of the handshaking of R is:

$$*[[\neg ri];\ [li];\ lo\uparrow;\ ro\uparrow;\ [ri];\ [\neg li];\ lo\downarrow;\ ro\downarrow]. \tag{7}$$

This leads to the production-rule expansion:

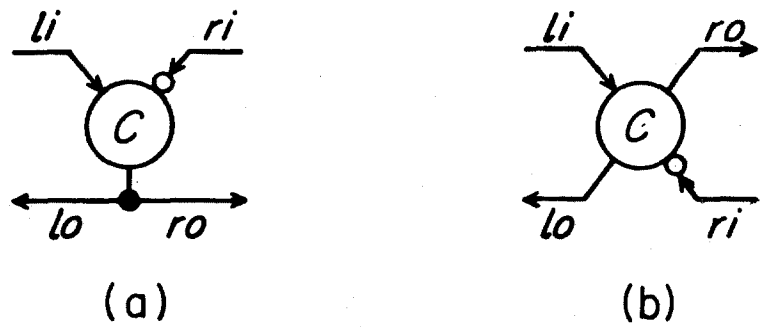
$$\neg ri \wedge li \mapsto lo\uparrow \tag{8}$$

$$lo \mapsto ro\uparrow\ \{\diamond ri \wedge \diamond \neg li\} \tag{9}$$

$$ri \wedge \neg li \mapsto lo\downarrow \tag{10}$$
$$\neg lo \mapsto ro\downarrow \{\diamond \neg ri\}. \tag{11}$$

Using the postconditions indicated between braces—these conditions rely on (5)—it is easy to verify that the production rules of the set are executed in program order. Hence the execution of the production-rule set is equivalent to the execution of (7).

The last step of the compilation, called *operator reduction*, consists in identifying production rules of the program with production rules defining the operators. We group the production rules that modify the same variable and we try to identify them with one or more operators. The production rules (8) and (10) are implemented as $(\neg ri, li)\ \underline{C}\ lo$. The production rules (9) and (11) are implemented as $lo\ \underline{w}\ ro$. The circuit is represented in Figure 1.a. with the alternative representation of Figure 1.b.



–Figure 1–

8. LESS SHUFFLING

Here, we shuffle only $lo\downarrow$ in the original handshaking expansion. We get:

$$*[[\neg ri \wedge li];\ lo\uparrow;\ [\neg li];\ ro\uparrow;\ [ri];\ lo\downarrow;\ ro\downarrow].$$

The production rule expansion gives:

$$\begin{aligned} \neg ri \wedge li &\mapsto lo\uparrow \\ lo \wedge \neg li &\mapsto ro\uparrow \\ ri &\mapsto lo\downarrow \\ \neg lo &\mapsto ro\downarrow . \end{aligned}$$

The two production rules that modify lo cannot be immediately identified with an operator, and the same for the two production rules that modify

ro. In such a case, we perform on the group a last transformation called *symmetrization*: we transform the guards of the production rules—again under invariance of the semantics—so as to make them "look like" the guards of operators. If the guard contains too many variables, this step may also involve decomposing a production rule into several ones by introducing additional variables called *padding variables*.

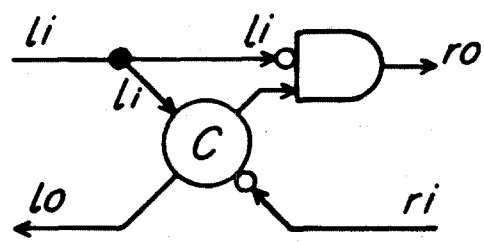
For the guards of lo, we observe that we can strengthen the guard ri of $lo\downarrow$ as $\neg li \wedge ri$ since $\neg li$ holds as a precondition of the production rule. For the guards of ro, symmetrization requires to weaken the guard $\neg lo$ of $ro\downarrow$ as $li \vee \neg lo$. In this case, since we have weakened the guard, we have to check that we have not enlarged the set of states in which the production rule can be effectively executed. Since $\neg ro$ holds when li holds, no such state has been added. Hence the transformation is safe. After symmetrization, we get the equivalent set:

$$\neg ri \wedge li \mapsto lo\uparrow \tag{12}$$
$$lo \wedge \neg li \mapsto ro\uparrow \tag{13}$$
$$\neg li \wedge ri \mapsto lo\downarrow \tag{14}$$
$$li \vee \neg lo \mapsto ro\downarrow . \tag{15}$$

Now, the operator reduction is straightforward:

$$(12)\&(14) : (\neg ri, li) \ \underline{C} \ lo$$
$$(13)\&(15) : (lo, \neg li) \ \underline{\wedge} \ ro.$$

which gives the circuit of Figure 2.



–Figure 2–

9. LESSER SHUFFLING

In this case we postpone the sequence $[\neg li]; lo\downarrow$ only until after $ro\uparrow$ and $[\neg ri]$ until after $lo\downarrow$. Again these shufflings maintain the cyclic order among the handshaking actions of L and among the handshaking actions of R. We get the program:

$$*[[li];\ lo\uparrow;\ [\neg ri];\ ro\uparrow;\ [\neg li];\ lo\downarrow;\ [ri];\ ro\downarrow].$$

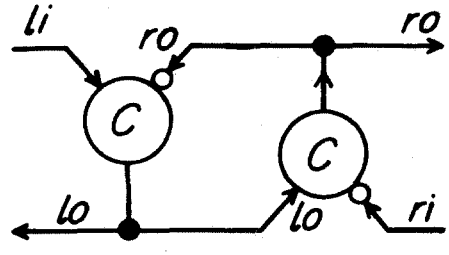
The production rule expansion is straightforward:

$$\begin{aligned}
&\neg ro \wedge li \mapsto lo\uparrow \\
&lo \wedge \neg ri \mapsto ro\uparrow \\
&\neg li \wedge ro \mapsto lo\downarrow \\
&ri \wedge \neg lo \mapsto ro\downarrow .
\end{aligned}$$

Which immediately leads to the operators:

$$\begin{aligned}
&(\neg ro, li)\ \underline{C}\ lo \\
&(lo, \neg ri)\ \underline{C}\ ro.
\end{aligned}$$

The circuit is represented in Figure 3.



–Figure 3–

10. QUICK-RETURN LINKAGE

We will now compile p without shuffling actions. We will observe that the compilation is more complicated than with shuffling but leads to more efficient circuits: the L-handshaking sequence is completed before the R-handshaking sequence starts. For this reason, such an implementation is sometimes called a "quick-return linkage"[8].

First implementation

(This solution has been designed together with Huub Schols, from Eindhoven University of Technology.) In order to define the precondition of $ro\uparrow$ uniquely, we now introduce a state variable u as follows:

$$*[[\neg ri \wedge li];\ lo\uparrow;\ u\uparrow;\ [u];\ [\neg li];\ lo\downarrow;\ ro\uparrow;\ [ri];\ u\downarrow;\ [\neg u];\ ro\downarrow\ ].$$

An additional problem here is that the condition $\neg ri \wedge li$ is not strong enough as precondition of $lo\uparrow$: since the implementation of L is passive, li

can become **true** after $lo\downarrow$, i.e. $\neg ri \wedge li$ may hold after $lo\downarrow$. We strengthen the condition with $\neg u$ and get the correct production rule set:

$$\neg ri \wedge li \wedge \neg u \mapsto lo\uparrow \quad (18)$$
$$lo \mapsto u\uparrow \quad (19)$$
$$\neg li \wedge u \mapsto lo\downarrow \quad (20)$$
$$u \wedge \neg lo \mapsto ro\uparrow \quad (21)$$
$$ri \mapsto u\downarrow \quad (22)$$
$$\neg u \mapsto ro\downarrow. \quad (23)$$

The symmetrizations of (19) & (22) and of (21) & (23) are straightforward:

$$\neg ri \wedge lo \mapsto u\uparrow \quad (19')$$
$$\neg lo \wedge ri \mapsto u\downarrow \quad (22')$$
$$u \wedge \neg lo \mapsto ro\uparrow \quad (21)$$
$$lo \vee \neg u \mapsto ro\downarrow. \quad (23')$$

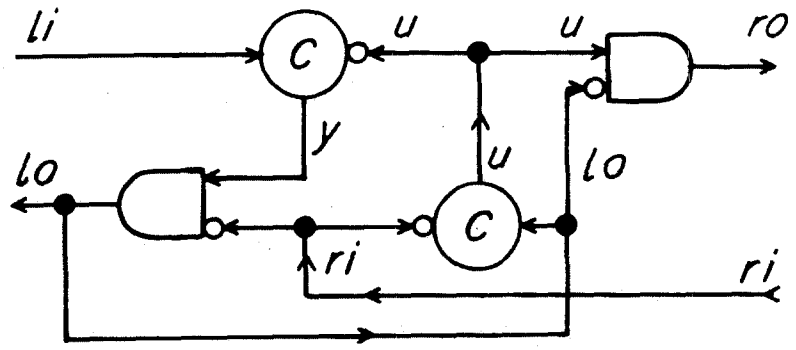
For the symmetrization of (18) & (20), since the guard of (18) contains three variables, we introduce a padding variable y to decompose the guard:

$$\neg u \wedge li \mapsto y\uparrow \quad (24)$$
$$\neg ri \wedge y \mapsto lo\uparrow \quad (25)$$
$$u \wedge \neg li \mapsto y\downarrow \quad (26)$$
$$ri \vee \neg y \mapsto lo\downarrow. \quad (27)$$

(For the newly introduced variable y, we have to check that no effective transition other than (24) and (26) is possible in the production rule expansion.) The operator reduction now gives:

$$(19') \ \& \ (22') \ : \ (\neg ri, lo) \ \underline{C} \ u$$
$$(21) \ \& \ (23') \ : \ (u, \neg lo) \ \underline{\Delta} \ ro$$
$$(24) \ \& \ (26) \ : \ (li, \neg u) \ \underline{C} \ y$$
$$(25) \ \& \ (27) \ : \ (\neg ri, y) \ \underline{\Delta} \ lo$$

which gives the circuit of Figure 4.



–Figure 4–

Second implementation

We decompose p into two processes q and t:

$$q \equiv *[C; R]$$
$$t \equiv *[[\overline{D} \rightarrow L; D]],$$

where (C, D) is a newly introduced channel. According to the process decomposition rule, $(q \,\|\, t)$ is equivalent to p. Because C matches D and D is probed, C has to be implemented as active and D as passive. It turns out that the implementation of q with C and R both active is simpler than the original one, and that t is also easy to implement. Again the compilation of q without shuffling requires introducing a state variable u:

$$q \equiv *[co\uparrow;\ [ci];\ u\uparrow;\ [u];\ co\downarrow;\ [\neg ci];\ ro\uparrow;\ [ri];\ u\downarrow;\ [\neg u];\ ro\downarrow;\ [\neg ri]].$$

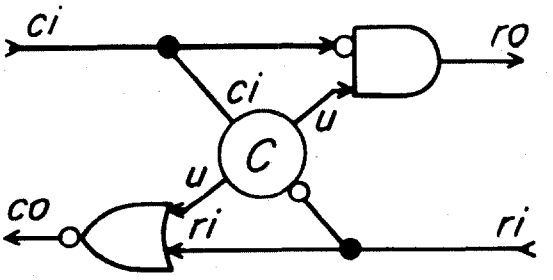
The production rule expansion gives:

$$\begin{aligned} \neg ri \wedge \neg u &\mapsto co\uparrow \\ \neg ri \wedge ci &\mapsto u\uparrow \\ ri \vee u &\mapsto co\downarrow \\ u \wedge \neg ci &\mapsto ro\uparrow \\ ri \wedge \neg ci &\mapsto u\downarrow \\ \neg u \vee ci &\mapsto ro\downarrow . \end{aligned}$$

The operator reduction gives:

$$\begin{aligned} (\neg ri, \neg u) &\,\underline{\wedge}\, co \\ (\neg ri, ci) &\,\underline{C}\, u \\ (u, \neg ci) &\,\underline{\wedge}\, ro. \end{aligned}$$

The circuit is shown in Figure 5, in which $(\neg ri, \neg u) \,\underline{\wedge}\, co$ is replaced by $(ri, u) \,\underline{\vee}\, \neg co$.



–Figure 5–

The compilation of t is straightforward. The handshaking expansion gives:

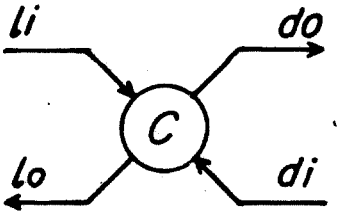
$$*[[di];\ [li];\ lo\uparrow;\ [\neg li];\ lo\downarrow;\ do\uparrow;\ [\neg di];\ do\downarrow].$$

Since D is an internal channel to t, we can shuffle the sequence $[\neg li]; lo\downarrow$ with respect to D without changing the order of L relative to R. We get:

$$*[[di];\ [li];\ lo\uparrow;\ do\uparrow;\ [\neg di];\ [\neg li];\ lo\downarrow;\ do\downarrow].$$

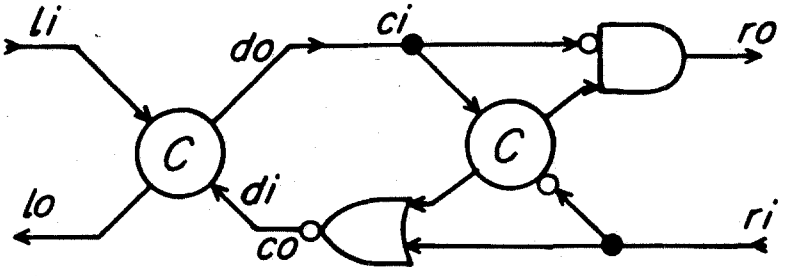
The production rule expansion leading to the circuit of Figure 6 is:

$$di \wedge li \mapsto lo\uparrow, do\uparrow$$
$$\neg di \wedge \neg li \mapsto lo\downarrow, do\downarrow.$$



–Figure 6–

The complete circuit of Figure 7 is obtained by composing the circuits of Figure 5 and Figure 6.



–Figure 7–

Let us now go back to the original program $p \equiv *[L?x;\ R!x]$ where x is an internal Boolean variable. In order to implement p, we duplicate the channels L and R and use channels $L1$ and $L2$ to input the values **true** and **false** respectively, and channels $R1$ and $R2$ to output the values **true** and **false** respectively. We get

$$\begin{aligned} pp \equiv *[[\ & \overline{L1} \rightarrow L1; R1 \\ | & \overline{L2} \rightarrow L2; R2 \\ ]], & \end{aligned}$$

with $\neg\overline{L1} \vee \neg\overline{L2}$ invariantly **true**. Using the first solution, we get the handshaking expansion:

$$\begin{aligned} pp \equiv *[[\ & \neg r1i \wedge l1i \rightarrow l1o\uparrow,\ r1o\uparrow;\ [r1i \wedge \neg l1i];\ l1o\downarrow,\ r1o\downarrow \\ | & \neg r2i \wedge l2i \rightarrow l2o\uparrow,\ r2o\uparrow;\ [r2i \wedge \neg l2i];\ l2o\downarrow,\ r2o\downarrow \\ ]]. & \end{aligned}$$

Next, we have to ensure mutual exclusion between the two guarded commands in order to be able to replace them by a set of production rules.

Assume pp is inside the first guarded command. Since $\neg l1i \vee \neg l2i$ holds as a consequence of $\neg\overline{L1} \vee \neg\overline{L2}$, the second guard is **false** as long as pp has not completed $l1o\downarrow$. Since $r1i$ holds until pp has completed $r1o\downarrow$, the second guard is guaranteed to remain **false** as long as pp is inside the first guarded command, if we strengthen the second guard as:

$$\neg r1i \wedge \neg r2i \wedge l2i.$$

And symmetrically for the first guard. We get:

$$\begin{aligned} pp \equiv *[[\ & \neg r1i \wedge \neg r2i \wedge l1i \rightarrow l1o\uparrow,\ r1o\uparrow;\ [r1i \wedge \neg l1i];\ l1o\downarrow,\ r1o\downarrow \\ | & \neg r2i \wedge \neg r1i \wedge l2i \rightarrow l2o\uparrow,\ r2o\uparrow;\ [r2i \wedge \neg l2i];\ l2o\downarrow,\ r2o\downarrow \\ ]]. & \end{aligned}$$

The production rule expansion for the first guarded command gives:

$$\neg r1i \wedge \neg r2i \mapsto a\downarrow \tag{28}$$

$$\neg a \wedge l1i \mapsto u\uparrow \tag{29}$$

$$u \mapsto l1o\uparrow, r1o\uparrow \tag{30}$$

$$ri1 \vee r2i \mapsto a\uparrow \tag{31}$$

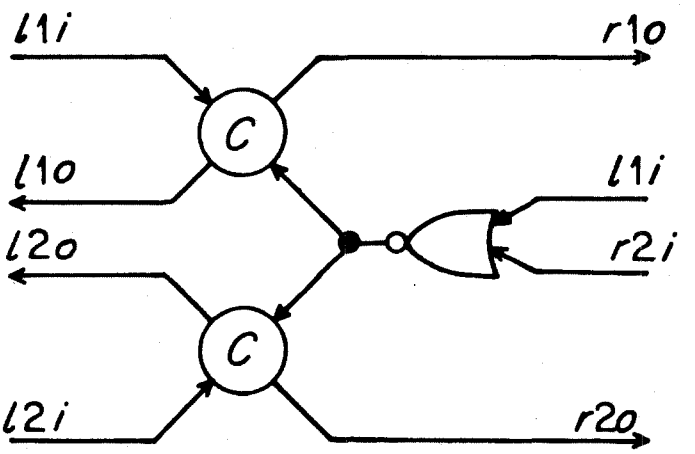
$$a \wedge \neg l1i \mapsto u\downarrow \tag{32}$$

$$\neg u \mapsto l1o\downarrow, r1o\downarrow \tag{33}$$

The operator reduction gives:

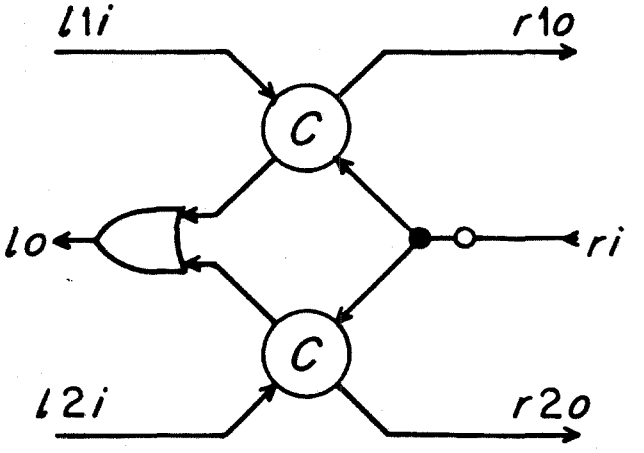
$$(28)\ \&\ (31) : (r1i, r2i)\ \underline{\vee}\ a$$
$$(29)\ \&\ (32) : (\neg a, l1i)\ \underline{C}\ u$$
$$(30)\ \&\ (33) : u\ \underline{f}\ (l1o, r1o)$$

The operator reduction of the second guarded command is identical. The final circuit is:



–Figure 8–

If we let the two L-channels share wire lo, and the two R-channels share wire ri, we get the circuit of Figure 9.



–Figure 9–

12. COMPLETE FIFO-ELEMENT WITH "QUICK RETURN"

We use the second "quick return" solution and decompose pp as:

$$q1 \equiv *[C1; R1]$$
$$q2 \equiv *[C2; R2]$$
$$t1 \equiv *[[\overline{D1} \wedge \overline{L1} \rightarrow L1; D1]]$$
$$t2 \equiv *[[\overline{D2} \wedge \overline{L2} \rightarrow L2; D2]]$$

In order to guarantee that the concurrent execution of $q1$, $q2$, $t1$, and $t2$ is equivalent to the execution of pp, we have to strengthen the guards of the handshaking expansions of $t1$ and $t2$ so as to enforce mutual exclusion between the executions of the first and the second guarded commands of pp. From the handshaking expansion of q and t in Section 10, we observe that when pp is executing its first guarded command, $l1i \vee \neg d1i$ holds, and symmetrically when pp is executing its second guarded command. Since $\neg l1i \wedge \neg l2i$ is guaranteed by definition, it suffices to strengthen the guards of $t1$ and $t2$ as $d1i \wedge l1i \wedge d2i$, and $d2i \wedge l2i \wedge d1i$, respectively.

Apart from this transformation, the rest of the compilation is identical to the compilation of q and t. The only difference, caused by the strengthening of the guards of $t1$ and $t2$ is that the production rules of $l1o$ in $t1$ have to be implemented by the enabled C-element:

$$(d1i, l1i; d2i)\ \underline{eC}\ l1o$$

and the production rules of $l2o$ in $t2$ have to be implemented by the enabled C-element:

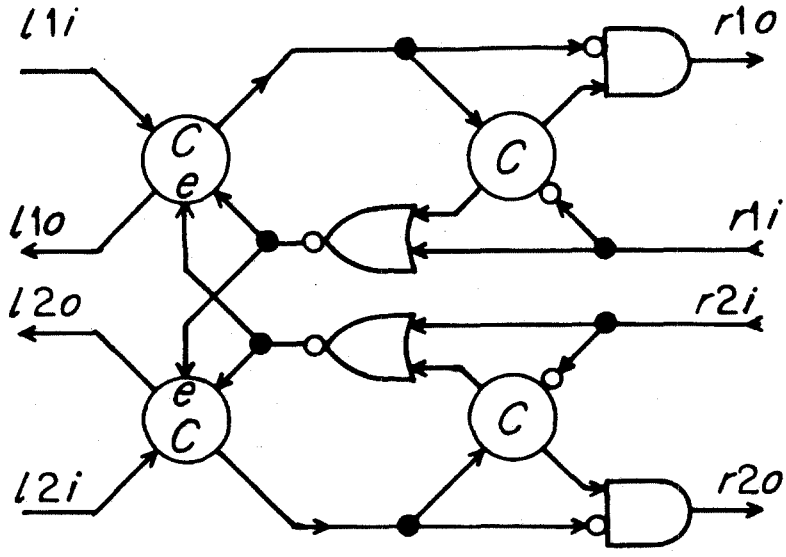
$$(d2i, l2i; d1i)\ \underline{eC}\ l2o.$$

The complete circuit is shown in Figure 10.

13. CONCLUSION

In this application of the method, we have shown how different circuits for a self-timed FIFO-element can be derived from the different heuristics that the method allows. The example also cleary illustrates the trade-offs between ease of compilation and simplicity of the circuits on the one hand, and efficiency on the other hand.

We have used only four-phase handshaking in this example, although two-phase handshaking is more efficient since it uses only half of the handshaking sequences. Unfortunately, two-phase handshaking is more difficult to realize because of the necessity to record the current value of the handshaking



–Figure 10–

variables, and therefore we have first developed a method based on four-phase handshaking. However, recent experiments with two-phase indicate that the method can handle both protocols

The operators used to construct the circuits are all well-known and VLSI implementations exist for all of them. (The enabled C-element is the only one the implementation of which is somewhat difficult. Fortunately, it can be replaced most of the time by an asymmetric C-element, which is easier to implement. This is the case for the circuit of Figure 10.)

The most important assumption on which the correct functioning of the circuits depends is the *stability* assumption for the guards of operators. The stability of a guard is guaranteed by two properties. One the one hand, the compilation method sees to it that a change of value on a single wire is followed by a change of value of the output variable of the operator the wire is an input of. A change of value on a fork is followed by a change of value of the output variable of at least one of the operators the fork is an input of. Since we assume the forks to be isochronic, this is enough to guarantee that the change has reached all outputs of the fork before a new change occurs. On the other hand, we may assume that a change of value–a change of voltage—on a VLSI wire is monotonic. The combination of these two properties guarantees the stability of the guards.

Often, the isochronicity of the forks is not necessary. When it is, it is enough to ensure, for a binary fork, that the delay in a branch of the fork is shorter than the delay in the gate to which the branch is *not* connected.

ACKNOWLEDGEMENTS

I would like to thank Dominique Borrione, Steve Burns, Pieter Hazewindus, Kevin Van Horn, and Martin Rem for their extensive comments on the manuscript, and David Black and Bob Sproull for several discussions over compiling self-timed FIFO. Calvin Jackson's expert assistance in the preparation of the manuscript was greatly appreciated.

REFERENCES

[1] Dijkstra, Edsger W., *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ (1976).

[2] Hoare, C.A.R., "Communicating Sequential Processes". *Comm. ACM* 21, 8, pp. 666–677 (August 1978).

[3] Martin, A.J., "The Probe: an Addition to Communication Primitives", *Information Processing letters* 20, pp. 125–130 (1985).

[4] Martin, A.J., "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion", *Proc. 1985 Chapel Hill Conference on VLSI*, ed. Henry Fuchs, pp. 247–260 (1985).

[5] Martin, A.J., "A Delay-Insensitive Fair Arbiter", Caltech Computer Science Technical Report 5193:TR:85 (1985).

[6] Martin, A.J., "Compiling Communicating Processes into Delay Insensitive VLSI circuits", to appear in *Distributed Computing*, vol. 1, no 3, 1986.

[7] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980).

[8] Seitz, C.L., "System Timing", Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980).

[9] van de Snepscheut, J., "Trace Theory and VLSI Design" LNCS 200, Springer-Verlag Berlin Heidelberg (1985).